A Modula-2 Cross-Compiler for the Macintosh

Honours Project
Computer Science Department
October, 1986

Jonathan Gifford

Supervisor : Rod Harries

Table of Contents

Table of Contents

Introduction - 1

There is a Modula-2 cross-compiler available on the Prime, which currently generates code for the Prime. This cross-compiler was developed at ETH (Eidgenoessische Teohnische Hochschule) in Zuerich, Switzerland, by a team lead by H. Seiler, and was based on the original compiler written by Niklaus Wirth. This cross-compiler is called Smiler-2, and has code generators for the MC-68000, MC6809, and the PDP-11. The code generation pass and linker for the Prime were written by Greg Ewing at the University of Canterbury in 1985.

The aims of this project were to translate the ETH MC-68000 code generator from CDC-6000 Pascal to Sheffield Pascal, and modify either the Prime or ETH linker so that it produced a Macintosh resource fork, which could then be incorporated into an application file on the Macintosh.

Modula-2 Programming - 2

2.1. Introduction

This section briefly outlines the nature of a Modula-2 program, and shows how this affects the compiler and linker. Modula-2 was designed to allow large or complex programs to be written as separate but quite closely connected modules, with a well-defined method of sharing any procedures, variables, constants or types between modules. The use of modules allows the programmer to decompose a problem, and write a single module to perform a specific task, to achieve the desired level of abstraction. Modula-2 programs either can be contained within a single source file, or they can be split into separate definition and implementation modules. Modules which are to be used by other programs are so split. The definition module contains all of the definitions that are available to users of that module.

2.2. Module life-span

Every imported module used in a program has a life-span equal to the running time of the main program. This means that all of the global variables within these modules must also have a life-span of this length, and so they are allocated room on the stack with the main program's globals. Each of the procedures within an imported module is treated exactly the same way as procedures within the main program - stack space for variables and parameters is allocated as required. Every module has a main body, within which it does whatever is required to ensure its exported entities are correct. Thus, the main body of all imported modules must be executed before the importer's main body, to ensure that the imports will be correct.

2.3. The SYSTEM and SYSTEMX modules

The Modula-2 report specifies that the compiler implementor should provide a pseudo-module called SYSTEM which provides a number of machine-specific facilities. These facilities should allow a programmer to gain access to the machine-language level of the machine, and they should be part of the compiler itself, rather than a separate module, but the facilities should be accessed as if they were in a module. There is also often a special module written in Modula-2 and called SYSTEMX which is used to support the code generated by the compiler and provides facilities such as floating point arithmetic, case statement handling, and various process related functions.

The Smiler-2 Cross-Compiler - 3

3.1. Structure

The Smiler-2 cross-compiler (as distributed) consisted of a number of passes, and a separate linker. These passes communicate using a number of inter-pass files, which contain pass-specific information about the program. The passes performed the following functions:

1. Syntax analysis
2. Declaration analysis
3. Body analysis
4. Code generation for expressions
5. Code generation for statements, and Link file generation
6. Symbol file generation
7. List file generation

The compiler available on the Prime had modified this structure slightly, by combining pass four and pass five into one pass, and incorporating the linker into the compiler itself. The MC-68000 code generator was split into a separate pass four and five, and this structure was maintained for this project.

3.2. Operation

The compiler will compile two different kinds of source files - module definition files (with a ".DEF" extension), and module implementation files (".MOD").

When a definition module is compiled the compiler will produce a symbol file (".SYM") which will contain details of all of the entities (procedures, variables, constants, and types) exported by that module.

When an implementation module is compiled the compiler will

produce a link file (".LNK") which contains all of the MC-68000 code for the module, as well as information for the linker. The creation of this file will entail the reading of symbol files associated with the modules from which this implementation module imports entities. Usually, this will involve (at least) its own symbol file (if it has one) and the SYSTEMX symbol file, which contains system specific data and procedures. Also generated at this stage is a reference file (".REF"), which contains some debugging information - this is not used in the current version of the compiler, but may be useful at some later date.

3.3. Inter-pass communication

The compiler communicates between passes both by means of internal data structures, and inter-pass files. These files contain a representation of the source file, which is composed of symbol tokens, and pointers to the internal data structures. As each pass reads the file, it is able to determine the information it requires to fulfill its functions, and generate the next inter-pass file. The internal data structures represent either a type or an identifier. A type record contains information about a particular type's size, kind, and composition, while an identifier record contains information about the identifier's location and class, and a pointer to its type. A dumper is available to dump the inter-pass file and internal structures, and may be used after any pass to verify that the pass is functioning correctly.

The Macintosh - 4

4.1. Introduction

The Macintosh provides a number of facilities which must be taken into account in the process of generating code. The most important of these is the memory manager which impacts both the code generation and the linking processes. Closely related to this is the idea of a resource, and the resource fork format, which has a profound effect on the activity of the linker. Next most important is the availability of the toolbox routines, for both the user and the implementor.

4.2. The Memory Manager

The Macintosh memory manager is quite simple in theory, and almost entirely automatic as far as an application is concerned. Space for the code of an application is allocated on the heap as required, with the heap growing from the bottom of memory (ie: lowest addresses) towards the top. Blocks of up to 32 kbytes may be allocated on the heap. The stack is used for storage of all application variables, and grows from the top of memory towards the bottom. The memory manager detects collisions between the stack and the heap, and handles garbage collection and heap compaction (when new heap space is required). The 32 kbyte limit for heap blocks means that programs which exceed this size must be split up into segments. It is possible to get around this limit, but for the sake of simplicity and convention, it was felt that this was an acceptable limitation.

4.3. The Jump Table

An application may be split into resource segments, which contain code and any fixed string data. These are loaded onto the heap as they are needed and removed if the space they occupy is required for other purposes. It is possible to access code in other segments using a jump table (which resides on the stack) and it

is through this structure that the loading of required segments is performed. When a routine in another segment is required, the calling code simply executes a jump to subroutine instruction into the appropriate part of the jump table, which executes code to either load the segment containing the routine onto the heap if it is not there already, or jumps directly to the routine if it is. At the start of program execution, the jump table consists entirely of code to load segments, and as segments are loaded the code in their associated jump table entries is modified from a load to a jump instruction. When segments are removed from the heap, their table entries are reset to the load instruction. This guarantees that a call to a routine in any segment will be successful.

a. Unloaded state

Offset from beginning of segment (2 bytes)
Machine instruction to push segment number onto the stack (4 bytes)
LoadSeg trap (2 bytes)

b. Loaded state

Segment number (2 bytes)
Machine instruction to jump to routine in memory (6 bytes)

The Jump table Entry Format

4.4. Resources

The idea of a resource is central to the Macintosh operating system, and the previous section shows how they are used with respect to application code. The task of the linker is to generate a "resource fork", containing the program code and fixed strings, which can be transferred to the Mac and then transformed into an application resource which can be used. A description of a

resource fork can be found in Appendix A. The task of transforming the output of the linker into a useable file is left to Rmaker on the Mac. It would have been possible to use the linker to generate a Macintosh application directly on the Prime, but it was felt that this would not give as much flexibility as the use of Rmaker, as well as being less productive than other work (since Rmaker could do this already).

4.5. The Toolbox

The Macintosh toolbox provides a very large number of facilities which allow the programmer to fully utilise the Mac operating system and user interface. Many of these facilities should be available directly to the compiler implementor when the system-specific sections of the compiler are being written. There are a large number of possible toolbox calls, with varying numbers of parameters, and two different parameter passing mechanisms (stack and register). The facilities provided include heap, resource, and process management, text and graphics manipulation, file system access, and the full range of user-interface tools. Obviously, the final Modula-2 compiler must be able to provide access to these facilities.

Design Decisions - 5

5.1. Code Generation

The code-generation passes from ETH were assumed to generate code correctly, and so there was really very little to be done directly as regards the code-generation phase of the compiler apart from getting the passes running. However, because the first three passes had been altered in order to get the Prime code-generation working, there were a number of changes to be made to pass four and five to fit in with these alterations. These changes, although they appeared relatively minor, in fact represented most of the work involved in getting passes four and five running. Other complications were encountered because of some of the non-standard features of the CDC-6000 Pascal, such as the ability to ORD a pointer to get an absolute address, and the CARDing of a set to determine the number of elements that are contained in it. The time spent on these problems was disproportionately large when compared to their effect on the final result, but there was no way to proceed while they were unsolved.

The only major change was to make the code position-independent, so that it would work with the Macintosh memory management system. The code originally produced relied upon being in a fixed location in the address space of the host system, but since the code must be placed on the heap in the Mac, and since the exact location of the code in the heap at run-time could not be known, it must be position independent. This was achieved by changing all absolute addressing mode references to data and code to PC relative mode references.

5.2 The Linker

There were three options in the choice of a linker to generate the Mac resource fork. It was possible to modify either the ETH linker or the Prime linker, or to write a completely new linker. The third of these options was ruled out almost immediately, since there was not time, and the result would probably be much the same as the Prime linker. The ETH linker was considered briefly, but its size and complexity meant that it was not practical to use it. If it had been chosen, the same conversion process from CDC Pascal to Sheffield Pascal would have to have been undertaken, involving about the same amount of work as the two code generation passes combined. The linker itself was also far too complicated, as it generated overlaid programs by inserting the appropriate overlaying code into the program, which is unnecessary for the Mac. The Prime linker, on the other hand, was relatively easy to alter to perform the necessary tasks. It had no overlaying facility, since the Prime is a virtual memory based machine, it was only about one tenth the size of the ETH linker, and it was written in Sheffield Pascal.

5.3. The Toolbox

The toolbox provides access to a large number of powerful facilities on the Macintosh. First thoughts on how to actually use these facilities centred around the modification of the skeleton UNIX call procedure already existing within the compiler. Two main factors prevented this idea from being used, the first of which is the sheer complexity of the task. There are several hundred toolbox calls available, and very few of these have the same parameter requirements (ie: the same number and type of parameters). Also, some of the routines require their parameters to be in registers, while other require them to be on the stack. Either the compiler would need a large amount of information on each toolbox call, or this information

would need to be incorporated into the call itself by the programmer. Neither of these solutions is very satisfactory. The second main factor was that there would need to be quite radical (though localised) changes to the compiler, in a number of passes, thus adding even more to the complexity of the compiler. These two factors effectively ruled out this option.

The solution adopted was to write the interface as a number of modules each of which contained the appropriate code to set up the parameters for the routines required. This was all done at the machine-language level within the Modula-2 framework, and the result was a very robust interface which was not in any way reliant on the compiler, and which involved no extra complexity within the compiler.

Code Generation - 6

6.1. Introduction

This section outlines the way in which Pass 4 and Pass 5 work, and gives an assessment of the code produced. Because these two passes were simply converted to Sheffield Pascal, the exact code they produce in all cases is still not fully known. An assessment is given of the code which has been generated, and suggestions are made to improve this code. In general, the code produced has been reasonably good.

6.2. Variable (and Constant) access

Variables can be accessed in one of three ways. If they are global, then they are accessed using an offset (into the global data block) from A5. This data block must use A5 as its base address, since this is the register around which the Macintoshes "A5 World" is built. The jump table, application parameters, and quickdraw globals all rely on this register being used. This caused some problems when any access is made to the current process descriptor block, since this is also pointed to by A5. Of the two, it was easier to change the process descriptor pointer to A4, as every reference to a Quickdraw global would have required that the current process descriptor be saved, the access made, and the process descriptor restored. Obviously, this is unsatisfactory. It was felt that the compiler should generate code as close to the Mac conventions as possible so A4 was used as the process descriptor pointer. This involved a rewrite of the ETH version of SYSTEMX, since it used A5 extensively, often embedded within constants with no mention of the fact!

If a variable is local, then it is accessed using an offset from the link register. The link register (MP) is set with the LINK instruction which saves the MP and SP of the active routine on

the stack, reserves a block of memory for local variables, and sets the MP and SP for the new routine. This instruction is used immediately upon entry to the new routine. Any parameters passed to the procedure are accessible with positive offsets from the current MP, and local variables have negative offsets.

Variables which are local to a procedure lower on the static link chain are accessed using a "Dijkstra display", in the process descriptor. This display contains the link register contents for all of the lower level procedures. This display is set up by pushing the link register contents into the appropriate place in the table (and every routine has a fixed static position within the program) whenever the routine is entered. (Only those routines which can be called by a higher level routine ever actually do this, so the highest level routines do not. The main body of the program never needs to do this either, since all of its variables are global) In this way, there will always be a valid static link chain from the current level to the main body of the program, and any routine can be sure that it can access any variables at a lower level with just one extra level of indirection. The use of this mechanism means that there is no need to store the static link chain on the stack, simplifying the stack frame, and speeding up access to lower level variables. There is a limit (of nine) to the number of static links that may be stored in this display, but it is unlikely that this limit will ever prove to be a serious constraint.

6.3. Pass 4 - Expression Code Generation

Pass 4 generates code for all expressions in the Modula-2 program (which includes all calls to user-defined or standard procedures and functions - the code for the latter is placed in-line, effectively making them macros). Each statement in the program is examined, and all of the expressions within each statement are evaluated, resulting in a block of code for each

expression. For example, if an assignment statement is being examined, then there must be an expression on the right hand side of the assignment symbol - the block of code generated will evaluate this expression. These blocks of code are inserted into the inter-pass file, to be used by pass5. As each block is generated, a number of link blocks are also created, which contain information for the linker about the references the newly generated code makes (to external, global or local data and procedures). When the linker reads these blocks, it will decide what action is necessary, and alter the existing code, or replace it with new code.

6.4. Pass 5 - Statement Code Generation

Pass 5 reads the inter-pass file, which now consists only of statements (in a tokenised form), and blocks of code from Pass 4 with their associated reference blocks. It generates code for the statements, including all procedure entries and exits, then combines this new code with that from pass 4 to give the final code for the program. This is written to the Link file, which contains a single block of code for each procedure, followed by the appropriate reference blocks from pass 4. The only major alteration to pass 5 was to change it from writing the code for individual procedures to the link file as it generated them to writing all of the code to the link file when the module had been completely compiled. This change was made so that the length of the module could be incorporated into the SCMODINIT block (Separately Compiled MODule INITialisation) at the start of the link file. This information is required by the Macintosh linker to enable it to allocate space in the code segments of the resulting resource file.

6.5. Code Quality

The code produced by the compiler is completely unoptimised (apart from constant expressions, which have been evaluated

by the compiler), but because of the nature of the MC68000 assembly language it is still quite compact (in terms of number of bytes per line of source). Making the code position independent resulted in shorter code, since a PC relative offset is limited to 16 bits, compared to a full 32 bits required for an absolute address. The use of a "Dijkstra display" results in compact code for access to all variables and constants. The compiler treats each identifier within an expression as a logically independent entity, and the code produced in evaluating expressions containing several identifiers at the same (non-local, non-global) lexical level contains some redundancy. This is because each time an identifier at this level is accessed, the compiler generates code to retrieve the static link from the display, then access the variable. When there are several different variables, then the display information is retrieved for each one, rather than just once. This could be improved by a more "intelligent" expression analyser - one which examined the lexical level of the operands, and performed this optimisation. The task of code optimisation is made more difficult because the compiler generates the binary machine code directly, rather than a text file containing assembler instructions. A "good" assembler would probably be able to optimise the code produced quite significantly.

The Linker - 7

7.1. Introduction

This section gives details on how the linker works, and what changes were made to the Prime linker to give a version which generated Macintosh resource files rather than Prime LOD files. All of the changes were made to a version of the linker which must still have been able to generate the correct Prime files, and this resulted in a size limitation, detailed later. It was necessary to examine the file produced by the linker to ensure that it really was a resource file, so a dumping program was written which accepted the linker output and produced a more readable result than that obtained by the FDUMP utility.

7.2. How the Linker works

The linker written by Greg Ewing last year uses four major data structures. The first of these is the **moddescriptor** array which contains information about each of the modules which is loaded. The seoond is the **procdescriptor** array, containing information on each of the procedures in the program. The third is the **reflist**, into which the reference blocks generated by Passes 4 and 5 are put. The last data structure is the **impdescriptor** array, which contains information about imported modules, relative to the importing module. For exact details on how these data structures work see Greg's report.
A (very) simple explanation of the linkers operation (for Prime programs) follows:

Δ The LNK file of the primary module is read. First, the imports are handled by reading a list of modules to import, and allocating each one an entry in the impdescriptor and moddescriptor arrays. Then the fixed string data is read

and put into the **code** array, and finally the code for each procedure body is read, along with the various reference blocks. As the code is read, it is put into the code array and an entry is put into the procdescriptor array. When the reference blocks are read, entries are added to the reflist array for the inter-module references, and all of the intra-module references are fixed immediately.

This same process (of reading imports, then strings, then code and reference blocks) is repeated for each of the imported modules, until there are no outstanding imports. At this stage, all of the code and fixed strings for the final executable image are available in the code array, with all of the inter-module links represented in the reflist.

Δ Each impdescriptor is then linked to the appropriate moddescriptor.

Δ The reflist is processed, and all inter-module references are fixed. This makes use of the imp/mod/procdescriptor arrays, and at the end of this processing, the code in the code array should represent the final executable code for the program.

Δ The main body of the module is built. This involves a call to each of the initialisation routines of each of the imported modules, in the reverse order to that with which they were loaded. This ensures that any calls to an imported module will function correctly, since the modules have all been initialised correctly before they are used.

Δ Finally, the LOD file is output.

7.3. What the new Linker does

The new linker, which generates Macintosh resource files as well as Prime LOD files, is a modification of the Prime linker in that it allocates space in the code array differently, modifies the moddescriptor entries, and creates a new data structure. It is implemented by simply modifying the source of the linker. In the following discussion the linker will be refered to as either the Mac or the Prime linker, when in fact it is one program - this is simply to make it clear that the two modes of operation are quite distinct.

The allocation of space in the code array is different for the Mac linker and the Prime linker because the Prime linker loads all of the code into a single segment of 128 kbytes as a contiguous block, while the Mac must fit as many whole modules as possible into segments of 32 kbytes. Because of the 128 kbyte limit for the Prime, the Mac linker is currently limited to only four segments, but this can be changed if required. For most purposes this should be ample.

The new data structure is the **jtable**, which contains information about the jump table entries. Each jump table entry, as described in 4.3, contains code which allows a procedure in one segment access to a procedure in another. The jtable array contains the information required to build this table; specifically, it has the offset into the jump table of the required procedure, and a linked list of locations within the code array of all of the places where this procedure is called. The array itself is built when the inter-module references are being fixed, since all of the modules have been loaded and it can be determined which of the calls are to procedures outside the segment in which the call originated.

Whenever there is a procedure call, pass 5 generates a PC

relative jump-to-subroutine instruction, and it is the task of the linker to find the correct offset and insert it after the JSR instruction - it is assumed that both the caller and the callee are in the same segment. If the call is to a procedure outside the current segment, then the linker must replace the PC relative instruction with an A5 relative one, and insert the correct offset into the jump table after it.

The modification to the moddescriptor entries is a field which contains the number of the chunk in which the code for that module appears. Each module must be contained completely within a single segment, since the fixed strings for the entire module are in a block at the start of the module. To give position independent code, and make it possible to reference these strings from anywhere within the module code, the code and strings must be in the same heap block. If they were not, it would be extremely difficult to try to determine exactly where the strings were on the heap at run-time. This information is used when jtable is being built. As each module is loaded, its length is added to the current pointer into the code array, and if the result is outside the current segment address space, the pointer is altered to point to the start of the next segment. It is not possible to overflow a segment with a single module, since pass 5 limits the size of a module to 32 kbytes.

7.4. The new main body

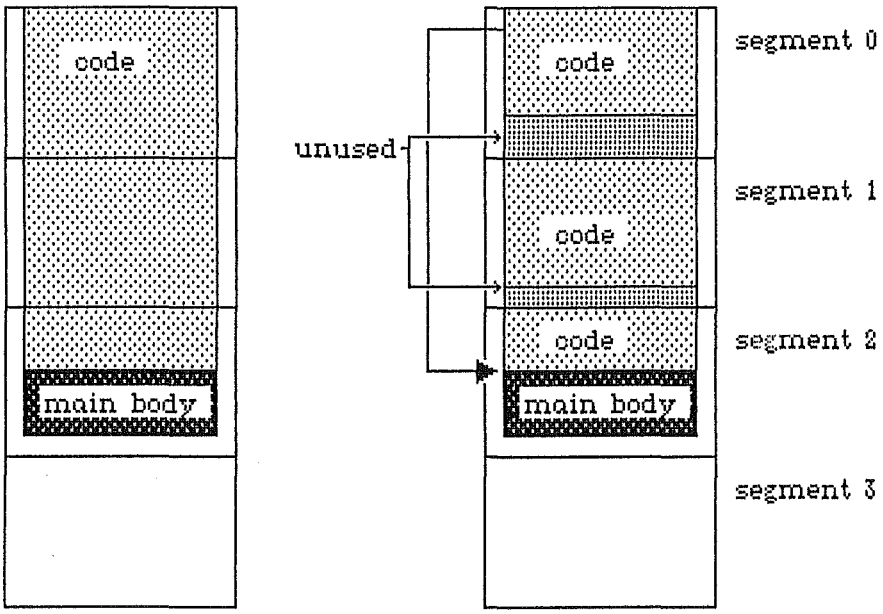
The main body of the resource file is different from the Prime version in that it contains calls to the Macintosh toolbox routines initgraf, initfonts, and initwindows. The main body provides the ideal location for these initialisation calls, since every module must use the QuickDraw routines to communicate with the user and the initgraf and initfonts calls must be made before this can happen. They must only be called once per program, and placing them in the main body frees the

programmer from the task of deciding which module to put them in to ensure they are called at the correct time.

A further reason for placing the initgraf call in the main body is that the argument it requires is a pointer to the QuickDraw globals. These globals should be stored as a block on the stack rather than on the heap - the Macintosh Revealed and Inside Mac books both recommend that they be on the stack, and this is the place one would expect global variables to be. The value of the pointer to these globals can only be known when all of the modules have been loaded and the total amount of stack space for the programs global data is known. If the initgraf call was placed inside a module, then there would have to be some way to ensure that the correct value was passed to the routine. This would complicate things for the linker, since it would need to know where in the code this pointer value should be inserted. Since the pointer is not a variable of any kind, but rather a linker dependent value, the added complexity required to allow the programmer access to this value was simply too great. Therefore, by making this call part of the main body of the program, the programmer is freed from trying to locate it properly, and the linker is freed from the complexities of inserting the value in the code.

Another consideration that only arises on the Macintosh is that the jump table must begin with an entry for the main entry point of the program. Because of the way the jump table is structured, this means that the main entry point must be in the first segment of the resource file. However, the main body of the program is generated after all of the modules have been loaded, and there is no way to tell how big it will be before they are all loaded. The combination of these two factors means that it is necessary to have a "dummy" main entry point at the very start of the program, which jumps to the "real" entry point of

the program (ie: the start of the newly created main body). The main body is treated as a separate module, and the jump can then be treated as an inter-module jump. If the entire program fits into a single segment then it is simply a PC relative jump, while if the program extends over multiple segments a jump table entry is required.

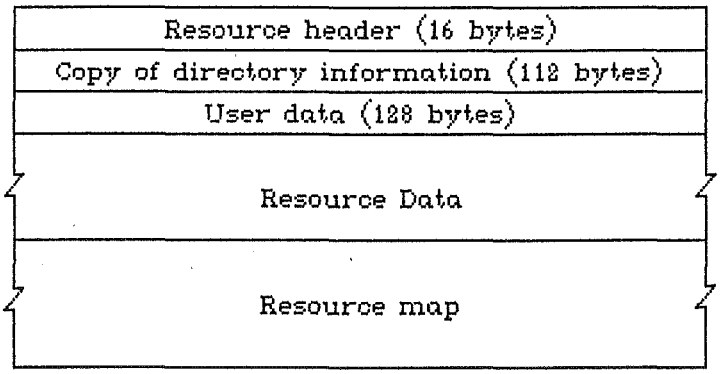


The **code** array contents

7.5. The Resource file

The output from the linker is a Macintosh resource file, which contains all of the executable code, and fixed string data. The implementation of this part of the linker was quite straightforward, since all of the information required had already been gathered into the various data structures. The file which is output is a binary data file, so the only way to check that it was correct was to dump it using the FDUMP utility in Primos, but there was no easy way to determine exactly what was what within the file. The solution to this problem was the dump program mentioned in the introduction to this section.

This program takes the binary input and splits it into the resource header, resource data, and resource map. The resource header contains information about the length of the data, and where the resource data and map are within the file. It also contains directory information, which is filled with nulls by the linker. The data consists of the code and strings, and "segment 0", which contains information for the resource manager about the jump table and size of global variables. The resource map is very simple since the linker only ever produces resources of type CODE.



Resource file format

Interfacing with the Toolbox - 8

8.1. Introduction

This section outlines the way in which the toolbox functions have been made available to the programmer. Throughout this section, "the toolbox" will refer to those routines described in the "Macintosh Revealed" book. Although this book leaves out some of the more low-level routines, it was felt that those that were described were sufficient for most applications. Section 5.3 describes why the interface was implemented as a number of modules, and this section goes into more details about the actual implementation of those modules. The books describe a Pascal based toolbox interface, but the Pascal and Modula-2 procedure calling and parameter passing techniques are so similar that it can safely be assumed that the Modula-2 interface will behave exactly as the Pascal one is claimed to.

8.2. The Parameter Passing schemes

There are two ways to pass parameters to the toolbox routines, the stack or the registers. In the first case, there are a number of conditions which apply to how the parameters actually appear on the stack. These are :

- Integers are two bytes long, long integers are four bytes long, and they are both in two's-complement form.
- All pointers are four bytes long.
- Booleans occupy two bytes on the stack, with the actual value in bit 8, the low order bit of the first byte.
- CHARs occupy two bytes, with the ASCII character code in the second byte.
- Character strings are represented by a pointer to the start of the string itself.

Δ If a data structure is more than four bytes long, then a pointer to it is passed, whereas any data structure less than four bytes long is stored directly onto the stack.

Δ All variable parameters are represented by a pointer to the actual data. This holds true whatever the type of the parameter.

These are the parameter passing conventions for stack-based toolbox calls. The Modula compiler has substantially similar conventions except for the following:

Δ All structured parameters, irrespective of size, are represented by their address.

Δ The ASCII code for a CHAR is placed in the first byte, not the second.

Δ Strings (in the Modula-2 sense) and dynamic arrays are represented by an address and length - since neither of these two types are used in the toolbox interface, this is not a problem.

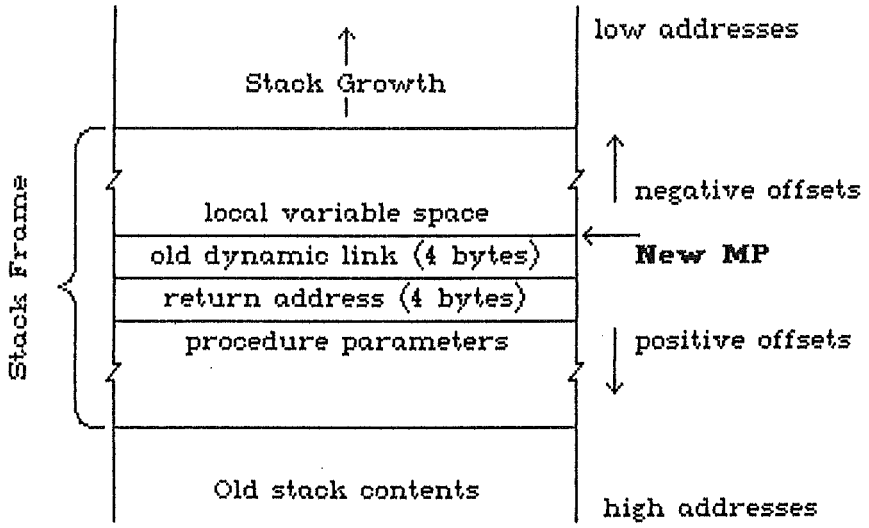
Notice that Pascal provides a long integer of 32 bits, whereas Modula-2 does not. Long integers are used quite frequently in the toolbox routines, so it was necessary to provide some sort of longint equivalent. This was done by defining a type **longint** to be an address, giving a longoard, in effect. The use of address as the base type had the advantage that the Modula compiler did not think the type was structured, and so it passed the actual value, rather than a pointer to the value. The problem with the CHAR's was easily resolved by popping bytes rather than words off the stack. The problem with structured types of four bytes or less was solved by retrieving the pointer and copying the value onto the stack, when required. All of the toolbox routines clean the stack up after they are finished, unlike the Modula-2 procedures which leave the stack clean-up to the calling

procedure.

The stack-based interface is generally much more complicated than the register based version. When the parameters are to be passed using the registers, the registers used are A0, A1, and D0. Registers A0 and D0 are always "free", and can be used however and whenever they are required. A1, on the other hand, should always be free, but it may be that it is not - to cater for this eventuality, all of the routines which use A1 first save it on the stack, and restore it before returning.

8.3. Implementation of the toolbox calls

When a procedure is called within a modula-2 program, the parameters are pushed onto the stack, then the JSR instruction is executed, pushing the return address onto the stack. Next, a LINK instruction is executed, which pushes the old dynamic link onto the stack and reserves space on the stack for all of the local variables and constants. After this, the stack looks as follows:



Thus, the effect of using Modula-2 interface routines is that all of the parameters are on the stack in the same format and order as they need to be for the toolbox call, with a few

exceptions, noted above. The implementation of the stack-based toolbox calls consisted in copying the parameters down from their location in the stack frame (modifying those which are not passed "correctly") into the stack-growth area, then executing the toolbox trap. This solution is not totally satisfactory, since in most cases it means copying the entire parameter block with no changes, when it is possible (with a little tweaking of the stack contents and pointers) to execute the toolbox trap directly. Because parameters are always passed on the stack ,the register based routines could not take advantage of this "loop-hole", so it was decided that for the sake of simplicity and uniformity all of the stack-based routines would use the parameter copying method, at least in the initial version of the toolbox interface. All of the code for the interface modules was written in MC68000 machine code, making use of the SYSTEM procedure CODE, and a large number of constants corresponding to the instruction set of the machine.

8.4. Stack based toolbox calls

As an example of the stack-based calls, consider the following procedure from the "menu" module:

```
PROCEDURE opendeskacc
          ( accname : str255 )
          : INTEGER;
BEGIN
   CODE( MKRMW,                    (1)
         MVLMPOMSP, 8,             (2)
         MTTopendeskacc,           (3)
         MVWSPPMPO, 12,            (4)
         UNLK, RTS);               (5)
```

The toolbox routine "opendeskacc" is passed the name of a desk accessory to open, and returns a reference number for it. The first line of the code above makes room on the stack for a result of size word (2 bytes). The code in line two moves a long word (4 bytes) starting at the location addressed by the offset 8 from the current value of the MP register onto the stack. Line 3 is the trap word for the toolbox routine - it is an unimplemented instruction which the Mac traps through a vector table at the start of physical memory. Line 4 pops a word from the stack, and puts it in the word before the long word parameter. The final line performs the unlink instruction, and returns from the routine.

This example shows two more important features of the use of Modula-2 procedures in the implementation of the toolbox. The first is that those routines which return a value need special code to actually return the value (on line 4) and also terminate properly (on line 5). Normally, when there is a result to be returned from a Modula-2 procedure, the RETURN statement is used. In the case of the toolbox routines, the use of the RETURN would have increased the complexity of the code, since the result would have had to have been placed in a register, then the RETURN would have to return the contents of that register. The direct approach was more efficient, although arguably less obvious. A compiler switch is available which allows the programmer to tell the compiler not to generate error code if a procedure which should return a value does not. When this occurrs the compiler generates code to trap to a SYSTEMX halt routine. This switch (the "R" switch) must be set off during compilation of the toolbox, since RETURN is never used. The compiler will not know that the value has been properly returned, and would generate this trap code, which is not desirable. The need for specific UNLK and RTS instructions is another by-product of not using the RETURN statement, since

these would have been generated by this statement.

The second feature is that the compiler does not generate code to copy the str255 parameter into the local variable space of the stack frame. For all non-toolbox routines, all non-variable parameters whose value can not be passed on the stack are copied into the local variable space, so that they can be used by the programmer in any way she desires. Since the toolbox routines are only an interface (or "glue") between Modula-2 and the Macintosh, there is never any need to modify any of these parameters, so they do not need to be copied. There is also a significant time penalty to be paid when the parameters to be copied are large (256 bytes in this case), which is completely unnecessary. These two factors forced the addition of a compilation switch ("T") to allow the programmer to select whether or not local copies are made. All toolbox modules are compiled with this switch turned on.

8.5. Register based toolbox calls

Consider the following procedure, as an example of a register based toolbox call:

```
PROCEDURE newhandle
            ( blocksize : size )
            : handle;
BEGIN
    CODE( MVLMPOD0D, 8,                (1)
          MTTnewhandle,                (2)
          MVLA0DMPO, 12,               (3)
          UNLK, RTS);                  (4)
```

The above is an example of a register based routine from the STORAGE module, which allocates a block of the size specified by the longword argument "blocksize", and returns a handle to the block. The block size is passed in register D0, and the value of the handle is returned in A0. In the code above, line 1 copies the blocksize from the parameter list into D0, line 2 executes the

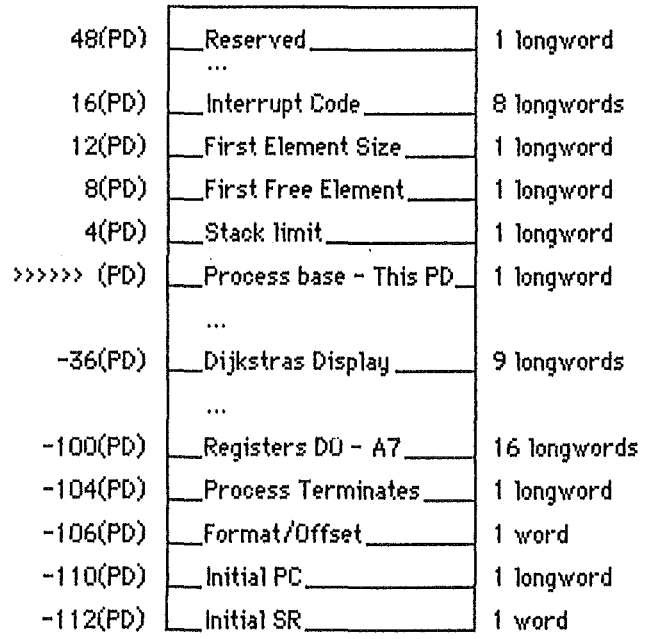
trap, line 3 copies the contents of register A0 into the space on the stack reserved for the procedure result, and line 4 unlinks and returns. For register based routines there is obviously no need to copy parameter blocks down the stack, and there is no need to make room on the stack for the result. The remarks made above about the use of the RETURN statement are not as valid, since the result will already be in a register, and the use of a RETURN would actually result in simpler code. Again, however, the first version of the toolbox interface was written to be as consistent as possible, so the same explicit code was used here.

8.6. The impact of the toolbox on SYSTEMX

The SYSTEMX which came with the compiler contained a number of procedures which duplicated toolbox activities, or Macintosh operating system operation. The most obvious of these was the stacktest procedure which was called at the start of each procedure to ensure that there was enough room on the stack for the local variables of that procedure. On the Macintosh, this is accomplished automatically by a collision detection mechanism that checks that the stack has not collided with the heap zone. Other procedures in SYSTEMX take care of CASE statement dispatching, process creation, process and I/O transfers, system calls, and halting. Of these, system calls are redundant since the toolbox interface provides all of the system facilities, halting could be done using the toolbox routines but wasn't for the sake of simplicity, and the process and I/O facilities are not currently implemented. The remainder of SYSTEMX consisted of arithmetic procedures for floating point and 32 bit integers and cardinals. There are standard packages available on the Mac for floating point arithmetic and transcendental functions on floating point numbers, but due to lack of time the first of these was not explored as a possible alternative to the SYSTEMX routines. The difference in

parameter passing mechanisms was also a factor, as the compiler generates code to jump to routines which simply set up the appropriate registers, then jumps to the routine which do the actual calculation.

The other alteration to SYSTEMX made necessary by the shift to the Macintosh was that the process descriptor block (one of which is created for each running process) was moved off the stack and onto the heap. A description of a process descriptor block follows:



Moving from the stack to the heap was necessary firstly because of the difficulty of actually finding a place on the stack for each descriptor, and secondly because it gave more flexibility. If the process descriptor is on the heap, then it will remain intact no matter what happens to the stack. The main function of the SYSTEMX initialisation code is to set up the process descriptor, and when it is complete register A4 points into the process descriptor. The use of the Macintosh heap/stack collision detection facility also meant that one of the primary roles of the process descriptor was no longer necessary - in a

"normal" MC68000 system, it was used to define the stack limit, since it was placed between the stack and heap. Any attempt to expand the stack beyond the limit set in the descriptor would result in the descriptor itself being overwritten. Most of the fields in the descriptor are now unnecessary, and later versions of SYSTEMX may address the problem of a new smaller block.

Conclusion - 9

Because it was impossible to test any of the code generated by the compiler until the linker was complete, and a copy of the resource file could be moved from the Prime to a Macintosh, the conclusions drawn must be tentative. Pass 4 and 5 appear to be functioning correctly, generating runnable but not spectacular code. The linker produces what appears to be a resource file, but there may need to be some tweaking of this file to convince a Mac that it is in fact a valid file (there are some fields in the resource header and map which contain flags pertaining to various aspects of the files behaviour). The actual task of getting a resource file from the Prime to a Mac seems to have boiled down to writing a slow transfer program from the Prime to the Vax. This program would need to convert the binary file into "inoffensive ASCII" in some way, and then transfer the file across the link with the Vax. A similar decoding program would be needed on the Vax side of the link, and from the Vax the file could be transfered directly to a Mac (using Macget, probably). Once the file is on the Mac, it should look like a resource file, and could then be massaged into an application file with Rmaker. This process should be quite straightforward - time was the major factor that prevented it from being completed.

The compiler does not currently support REALs, the CASE statement, or process creation and switching. The implementation of REALs should be quite straightforward, since all of the appropriate code exists in SYSTEMX for the arithmetic operations. The CASE statement should also be quite simple, since the current version (in SYSTEMX) is implemented using a table of PC-relative values to the appropriate code. Process creation and switching will be implemented when the rest of

the compiler is working properly. As with the CASE statement, all of the code for the process related activity is present in the ETH SYSTEMX, but will need to be rewritten to fit in with the Macintosh conventions. In particular, there will need to be a redefinition of a "process workspace", which is defined to be the available stack space for a process, and is delimited by the process descriptor. The process descriptor has been moved onto the heap, and the stacktest procedure has been discarded in favour of the Macintosh collision detection mechanism. This procedure can be reinstated, when and if it is necessary.

Appendix A

An example Modula-2 Program - The source code

TUE, 07 OCT 1986 test.mod

```
MODULE test;

VAR
  i     : INTEGER;

  PROCEDURE mary ();
  VAR
    j, k : INTEGER;

    PROCEDURE jane ();
    VAR
      l : INTEGER;
    BEGIN  (* jane *)
      i := 1;
      j := 2;
      k := j*2 + k DIV 5;
      l := i + j + k;
     END jane;

  BEGIN (* mary *)
    i := 1;
    j := 2;
    k := 3;
    jane ();
    END mary;

BEGIN (* main *)
  i := 12345;
  mary ();
  END test.
```

Appendix A

An example Modula-2 Program - The listing file

```
TUE, 07 OCT 1986                    1                    test.mod.list


Module  text from  TEST.MOD
Listing  to  file  TEST.MOD.LIST
Compiling  Module  test

      SMILER-2  -  MODULA-2 CROSS COMPILER VERSION 2.     04/10/86     17:42:27

    1         MODULE test;
    2
    3         VAR
    4           i     : INTEGER;
    5
    6           PROCEDURE mary ();
    7           VAR
    8             j, k : INTEGER;
    9
   10             PROCEDURE jane ();
   11             VAR
   12               l : INTEGER;
   13             BEGIN  (* jane *)

      0000: LINK A6,#$FFFE
   14               i := 1;

      0004: MOVE.W #0001,$FFFE(A5)
   15               j := 2;

      000A: MOVEA.L $FFFC(A4),A3  MOVE.W #0002,$FFFE(A3)
   16               k := j*2 + k DIV 5;

      0014: MOVEA.L $FFFC(A4),A3  MOVE.W $FFFE(A3),D5  ASL.W #1,D5
      001E: MOVEA.L $FFFC(A4),A3  MOVE.W $FFFC(A3),D4  EXTL D4  DIVS #0005,D4
      002C: ADD.W D4,D5  MOVEA.L $FFFC(A4),A3  MOVE.W D5,$FFFC(A3)
   17               l := i + j + k;

      0036: MOVEA.L $FFFC(A4),A3  MOVE.W $FFFE(A5),D5  ADD.W $FFFE(A3),D5
      0042: MOVEA.L $FFFC(A4),A3  ADD.W $FFFC(A3),D5  MOVE.W D5,$FFFE(A6)  UNLK A6
      0050: RTS
   18               END jane;
   19
   20           BEGIN (* mary *)

      0052: LINK A6,#$FFFC  MOVE.L $FFFC(A4),-(SP)  MOVE.L A6,$FFFC(A4)
   21               i := 1;

      005E: MOVE.W #0001,$FFFE(A5)
   22               j := 2;

      0064: MOVE.W #0002,$FFFE(A6)
   23               k := 3;

      006A: MOVE.W #0003,$FFFC(A6)
   24               jane ();

      0070: JSR 0000(PC)  MOVE.L (SP)+,$FFFC(A4)  UNLK A6
      007A: RTS
```

Appendix A

An example Modula-2 Program - The listing file

```
TUE, 07 OCT 1986                2                test.mod.list


    25              END mary;
    26
    27          BEGIN (* main *)

       007C: LINK A6,#0000
    28             i := 12345;

       0080: MOVE.W #$3039,$FFFE(A5)
    29             mary ();
    30             END test.

       0086: JSR 0000(PC)  UNLK A6
       008C: RTS
       008E: ORI.B #00,D0

       0092: ORI.B #00,D0
---SKIPPED LINES :


No errors detected
 Generated REF-file test.REF
 Generated LNK-file test.LNK
 Generated LOD-file test.LOD
```

Appendix A

An example Modula-2 Program - The Resource file (FDUMP)

TUE, 07 OCT 1986 1 test.res.octw

DUMP OF FILE:- TEST.RES

-WORD-	-------ASCII--------	----------OCTAL WORDS----------									
1	@@@@@@@h@@@h@@@@@@@@	000000	000400	000000	010350	000000	007750	000000	000034	000000	000000
11	@@@@@@@@@@@@@@@@@@@@	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000
121	@@@@@@@@@@@@@@@@@@@@	000000	000000	000000	000000	000000	000000	000000	000000	000000	000030
131	@@@@@@@@@@@@@@@@@@@@	000000	000050	000000	000160	000000	000010	000000	000040	000000	036474
141	@@)p@@@D@@@@@z@B@@@^	000000	124760	000000	007704	000000	000001	047372	007670	047126	177776
151	@@@@@^@@@!@@@@@^@@@!	035574	000001	177776	023154	177774	033574	000002	177776	023154	177774
161	@@@^c@@@@!@@@!@D@!@@	035053	177776	161505	023154	177774	034053	177774	044304	104774	000005
171	Z@@@@!@@@!@@@!@@@^Z@	155104	023154	177774	033505	177774	023154	177774	035055	177776	155153
181	@^@@@!Z@@!@@@^@@@@@@	177776	023154	177774	155153	177774	036505	177776	047136	047165	047126
191	@!@@@!@@@!@@@@@^@@@@	177774	027454	177774	024516	177774	035574	000001	177776	036574	000002
201	@^@@@@@!@:@@@@@!@@@@	177776	036574	000003	177774	047272	000000	024537	177774	047136	047165
211	@@@@@@@@@^@:@@@@@@@@	047126	000000	035574	030071	177776	047272	000000	047136	047165	020101
221	@@@@@@@@@@@@@@@@@@@@	052040	040504	042122	042523	051440	026455	026455	026455	026455	027000
231	@@@@@@@@@@@@@@@@@@@@	020114	044516	042440	030461	030461	020105	046525	046101	052117	051040
241	@@@@@@@@@@@@@@@@@@@@	024105	046461	030461	030440	052122	040520	177474	000000	020114	044516
251	@@@@@@@@@@@@@@@@@@@@	042440	030460	030460	020105	046525	046101	052117	051040	024105	046461
261	@@@@@@@@@@@@@@@@@@@@	030061	030040	052122	040520	024456	000000	020120	051111	053111	046105
271	@@@@@@@@@@@@@@@@ @@	043505	020126	044517	046101	052111	047516	020050	050122	177640	053111
281	@@@@@@@@@@@@@@@@@@@@	047440	052122	040520	024456	000000	020101	051111	052110	046505	052111
291	@@@@@@@@@@@@@@@@@@@@	041440	047526	042522	043114	047527	020050	052122	040520	053051	027000
301	@@@@@@@@@@@@@@@@@@@@	020111	047104	042530	027526	040514	052505	020117	052524	020117	043040
311	@@@@@@@@@@@@@@@@@@@@	051101	047107	042440	024103	044113	020111	047123	052122	052503	052111
321	@@@@@@@@@@@@@@@@@@@@	047516	024456	000000	020132	042522	047440	042111	053111	042105	020050
331	@@@@@@@@@@@@@@@@@@@@	055105	051104	044526	020124	051101	050051	027000	020111	046114	042507
341	@@@@@@@@@@@@@@@@@@@@	040514	020111	047123	052122	052503	052111	047516	020050	044514	046111
351	@@@@@@@@@@@@@@@@@@@@	047123	020124	051101	050051	027000	020101	042104	051105	051523	020105
361	@@@@@@@@@@@@@@@@@@@@	051122	047522	020050	040504	051105	051122	020124	051101	050051	027000
371	@@@@@@@@@@@@@@@@@@@@	020102	052523	020105	051122	047522	020050	041125	051505	051122	020124
381	@@@@@@@@@@@@@@@@@@@@	051101	050051	027000	020120	051117	043522	040515	020110	040514	052056
391	@@@@@@@@@@@@@@@@@@@@	000000	020116	047440	051105	052125	051116	020106	051117	046440	043125
401	@@@@@@@@@@@@@@@@@@@@	047103	052111	047516	020105	051122	047522	027000	020103	040523	042440
411	@@@@@@@@@@@@@@@@@@@@	044516	042105	054040	047525	052040	047506	020122	040516	043505	027000
421	@@@@@@@@@@@@@@@@@@@@	020123	052101	041513	020117	053105	051106	046117	053440	047503	041525
431	@@@@@@@@@@@@@@@@@@@@	051105	042056	000000	020111	047104	042530	027526	040514	052505	020117
441	@@@@@@@@@@@@@@@@@@@@	052524	020117	043040	051101	047107	042456	000000	020101	051111	052110
451	@@@@@@@@@@@@@@@@@@@@	046505	052111	041440	047526	042522	043114	047527	027000	020111	046520
461	@@@@@@@@@@@@@@@@@@@@	051117	050105	051040	053517	051113	051520	040503	042440	052117	020116
471	@@@@@@@@@@@@@@@@@@@@	042527	050122	047503	042523	051456	000000	020120	051117	041505	051523
481	@@@@@@@@@@@@@@@@@@@@	020124	042522	046511	047101	052105	042056	000000	020103	040514	046040
491	@@@@@@@@@@@@@@@@@@@@	052117	020125	047111	046520	046105	046505	047124	042504	020122	047525
501	@@@@@@@@@@@@@@@@@@@@	052111	047105	020111	047040	051531	051524	042515	054056	000000	020102
511	@@@@@@@@@@@@@@@@@@@@	051105	040513	027000	020110	042501	050040	047526	042522	043114	047527
521	@@@@@@@@@@@@@@@@@@@@	027000	020111	046114	042507	040514	020104	044523	050117	051505	027000
531	@@@@@@@@@@@@@@@@@@@@	020125	047113	047117	053516	020111	047104	042530	020124	047440	051531
541	@@@@@@@@@@@@@@@@@@@@	051524	042515	054056	044101	046124	052056	000000	047126	000000	020137
551	@@O(@@@@c@P(@@Pp@@@P	110220	130250	000004	061014	161600	150250	000010	150360	004000	047320
561	Qh@@@@@P@@@@@@@@@@@@	150750	000010	054610	047320	047136	047165	047126	000000	051617	017456
571	@@(@@@@@@@@^@@@^@@@^	000010	124203	047136	047165	047126	177776	041156	177776	035056	177776
581	:@@@@@@@@^@@@@@@@@@@	135156	000014	061046	035056	177776	023156	000010	045063	050400	063430
591	@@@^@@@@@@@@:@@@@@@	035056	177776	023156	000010	017463	050400	047272	000000	052217	051156
601	@^@P@@@@@:@@@@@@@@@:	177776	060720	017474	000012	047272	000000	052217	017474	000015	047272

Appendix A

An example Modula-2 Program - The Resource file (FDUMP)


```
  611  @@@@@@@@@@@@f@z@@@n@J  000000 052217 047136 047165 047126 177746 043772 000000 042756 177752
  621  @@@[@M@!@2@@@f@@@f@@@  075025 012333 053715 177774 177662 000024 177746 051556 177746 035056
  631  @@@@@"@@@h@@@@@@@@@@@  000010 074017 177642 036505 177750 035056 000010 177630 036505 000010
  641  @@@h@@@@@@@@@h@@@@@@@  035056 177750 006105 000012 062020 035056 177750 003105 000060 036505
  651  @h@z@@@@@h@@@@@@@h@@@  177750 047372 000016 035056 177750 003105 000067 036505 177750 035056
  661  @h@@@f@n@J@@@@@@@f@@@  177750 034056 177746 043756 177752 013605 040400 035056 177746 006105
  671  @@@@@$@@@@@@@ !@:@@@@  000014 063402 060644 037474 000025 044156 177752 047272 000000 056217
  681  @@@@@@@!@@@@@!@@@@@@@  047136 047165 047126 177774 000174 003400 044354 177777 177634 036500
  691  @!@@@@@@@z@@@@@!@@@w  177774 075400 045005 063404 047372 001130 035056 177774 006105 177767
  701  @@@@@@@@@@@:@@@@@z@@  063022 037474 000041 044172 000026 047272 000000 056217 047372 001074
  711  @@@!@@@x@@@@@@@@@@@:  035056 177774 006105 177770 063022 037474 000041 044172 000072 047272
  721  @@@@@z@@@@@!@@@y@@@@  000000 056217 047372 001040 035056 177774 006105 177771 063022 037474
  731  @@@@@@@:@@@@@z@@@@@!  000043 044172 000136 047272 000000 056217 047372 001004 035056 177774
  741  @@@z@@@@@@^`@@@:@@@@  006105 177772 063022 037474 000034 177340 000204 047272 000000 056217
  751  @z@h@@@!@@@{@@@@@@@@@  047372 000750 035056 177774 006105 177773 063022 037474 000053 044172
  761  @"@:@@@@@z@L@@@!@@@!  000242 047272 000000 056217 047372 000714 035056 177774 006105 177774
  771  @@@@@@@@@P@:@@@@@z@O  063022 037474 000032 044172 000320 047272 000000 056217 047372 000660
  781  @@@!@@@}@@@@@@@@@l@:  035056 177774 006105 177775 063022 037474 000042 044172 000354 047272
  791  @@@@@z@@@@@!@@@^@@@@  000000 056217 047372 000624 035056 177774 006105 177776 063022 037474
  801  @@@@@@@:@@@@@z@@@@@!  000034 044172 000420 047272 000000 056217 047372 000570 035056 177774
  811  @@@@@@@@@@@@@@@:@@@@  006105 177777 063022 037474 000030 044172 000456 047272 000000 056217
  821  @z@@@@@!@@@@@@@@@@:  047372 000534 045156 177774 063022 037474 000015 044172 000510 047272
  831  @@@@@z@@@@@!@@@@@@@@  000000 056217 047372 000504 035056 177774 006105 000001 063022 035144
  841  @@^@@@@:@@@@@z@@@@@!  000036 177040 000530 047272 000000 056217 047372 000450 035056 177774
  851  @@@@@@@@@@^@@@@:@@@@  006105 000002 063022 035110 000030 177004 000570 047272 000000 056217
  861  @z@@@@@!@@@@@@@@@@}h  047372 000414 035056 177774 006105 000003 063022 035054 000027 176750
  871  @@@:@@@@@z@p@@@!@@@@  000622 047272 000000 056217 047372 000360 035056 177774 006105 000004
  881  @@@@@@}L@,@:@@@@@z@T  063022 035020 000031 176714 000654 047272 000000 056217 047372 000324
  891  @@@!@@@@@@@t@@}O@H@:  035056 177774 006105 000005 063022 034764 000024 176660 000710 047272
  901  @@@@@z@B@@@!@@@@@@@X  000000 056217 047372 000270 035056 177774 006105 000006 063022 034730
  911  @@}@@^@:@@@@@z@@@@@!  000041 176624 000736 047272 000000 056217 047372 000234 035056 177774
  921  @@@@@@@<@@@}@@@@:@@@@  006105 000007 063022 034674 000023 176570 001002 047272 000000 056217
  931  @z@@@@@!@@@@@@@ @@}@  047372 000200 035056 177774 006105 000010 063022 034640 000051 176534
  941  @@@:@@@@@z@@@@@!@@@@  001030 047272 000000 056217 047372 000144 035056 177774 006105 000011
  951  @@@@@@}@@@@:@@@@@z@@  063022 034604 000006 176500 001104 047272 000000 056217 047372 000110
  961  @@@!@@z@@@}@@@@@@@@:  035056 177774 006105 175072 063022 176450 000016 044172 001114 047272
  971  @@@@@z@@@@@!@@z@@@}@  000000 056217 047372 000054 035056 177774 006105 175037 063022 176414
  981  @@@@@@@:@@@@@z@@@@@@  000020 044172 001134 047272 000000 056217 047372 000020 037474 000037
  991  @@!@@:!p@@@@@<@!@@@^  044172 176146 047272 176360 056217 026555 177674 177774 037456 177776
 1001  @:@@@@^@@@@@@!T@@@@  047272 000000 052217 060776 047136 044121 047126 176324 020127 047116
 1011  @@@@@@@@@@@h@@@@!1@@  047136 047165 047126 000000 020117 040750 177600 110700 130754 000004
 1021  @@!B@@@@@@@@@@@@@@@@  066402 176270 070003 020127 047116 047136 047165 047126 000000 070007
 1031  @:@@@@@@@@!@@@@@@1@@  047272 000000 047136 047165 047126 176234 000174 003400 044354 000000
 1041  @@@@@z@b@@@@@!@@@L!@  177634 020014 044772 177742 021124 021200 046354 000000 177714 176200
 1051  @@@@@@@@@@@@@@@@@@@@  000000 047136 047165 047126 000000 035056 000020 004005 000000 063026
 1061  @@@@@@!@@@@@@@@@@@@@  035056 000016 004005 176144 063014 035056 000016 006105 000000 000400
 1071  @@@@@@@@@@@@@@!@@@@!  062010 070006 020156 000004 047116 020156 000024 176110 000020 110374
 1081  @@@@@@@@@@@@@@@@@@@h  000064 041220 020556 000024 000004 020574 177777 177777 000010 042750
 1091  @@!@@@@@@Y@!@@@@@@@H  000020 176054 000000 000000 032331 063374 021110 070155 041041 050710
 1101  @!@@@X@(@T!@@L@@@@@@  177774 020511 177730 041250 177724 176020 177714 022156 000014 022210
 1111  @@@@@@@@@@@@@@@@@A(t  022174 000000 000000 020512 177630 020556 000030 177624 040301 175764
 1121  @@@@@@@@@@x@@@@@(b@!  000010 063412 001100 003400 001101 174377 101100 030501 175742 020774
 1131  @@@@@0@!@@@@!@@@@@@@  000000 000000 000260 020774 000000 000000 176162 047136 047165 047126
 1141  @@@@@@@@@@@@@@@@@@@1  000000 047114 047165 047136 047165 047126 000000 000174 003400 044354
 1151  @@@L@@@@@@@!@@@@@@@@  000000 177714 021121 020214 024111 046354 000000 177634 047163 047136
```

Appendix A

An example Modula-2 Program - The Resource file (FDUMP)

TUE, 07 OCT 1986 3 test.res.octw

1161	@@@@@@@@@@@@@@@@@@@@	047165	047126	000000	047113	047165	047136	047165	047126	000000	000174
1171	@@@1@@@L@@@1@@@@@@@@	003400	044354	000000	177714	024211	043754	000020	022213	021121	020214
1181	@@@1@@@@@@@@@@@@@@!	024111	046354	000000	177634	047163	047136	047165	047126	000000	020774
1191	@@@@@4@@@@@@@@@@@@@@	000000	000000	000264	047115	047165	047136	047165	047126	000000	000174
1201	@@@@@@@@@@@@O@@@@@@@	003400	045000	063024	030027	001100	003400	130110	065010	030010	001127
1211	x@@@@@@@@@@@@@@@@@@!	174377	100527	047163	051400	063004	037210	047163	051400	063006	043374
1221	@@@@@@@@@@@@@@@@@@@@	020000	047163	051400	063004	047160	047163	047163	047136	047165	047126
1231	@@@g@@@@@@@@@@@@@@@@	000000	044347	000000	071000	031056	000010	072000	032056	000012	023001
1241	@@@@@@@@BBDDFEHE@@R@	024001	025002	044104	044105	141302	142304	143305	144305	044101	151102
1251	@@Y@R@Y@@@@@@@@@@@T@	041205	154605	151103	154605	044101	041102	041103	044102	044103	152203
1261	T@@@@@@@@@@_@@@@@@@@	152204	036501	000012	036502	000010	046337	000000	047136	047165	047126
1271	@@@g@@@@@@@@@@@@@@4<	000000	044347	000000	071000	031056	000012	072000	032056	000010	132274
1281	@@@@@@@@@B@@@@@@@@@B	000000	177777	061036	041103	101302	064016	034001	041101	044101	101302
1291	@@@@@B@@@@@@@@@@@@@@	033001	031004	101302	044103	033001	041101	044101	060036	041203	033001
1301	@@@@@@@@@@c@c@2@@@@@	044103	041101	044101	034074	000017	161613	161621	131202	062404	111202
1311	@@@L@r@@@@@@@@@_@@@@	051103	050714	177762	036503	000012	036501	000010	046337	000000	047136
1321	@@@@@@@g@@@@@@@@@@@@	047165	047126	000000	044347	000000	071000	031056	000010	072000	032056
1331	@@@@@@@@@@@@@@@@BBDD	000012	023001	024001	025002	026001	041207	044104	044105	141302	142304
1341	FEHE@@R@Y@R@Y@@@@@@@	143305	144305	044101	151102	154607	151103	154607	044101	041102	041103
1351	@@@@T@T@@@@@@@@@@@@@	044102	044103	152203	152204	044105	045206	065002	112205	045205	065002
1361	@@@@@@@@@@_@@@@@@@@	112206	036501	000012	036502	000010	046337	000000	047136	047165	047126
1371	@@@@@:@@@@@@@@@@g@@	000000	070010	047272	000000	047136	047165	047126	000000	044347	000000
1381	@@@@@@@@@:@@@@@@@@_	026056	000010	027056	000014	047272	000000	047166	026507	000014	046337
1391	@@@@@@@@@@g@@@@@@@@	000000	047136	047165	047126	000000	044347	000000	026056	000010	027056
1401	@@@:@@@@@@@_@@@@@@	000014	047272	000000	047166	026507	000014	046337	000000	047136	047165
1411	@@@@g@@@@@@@@@@:@@	047126	000000	044347	000000	026056	000010	027056	000014	047272	000000
1421	@@@@@@@_@@@@@@@@@@g	047166	026507	000014	046337	000000	047136	047165	047126	000000	044347
1431	@@@@@@@@@@:@@@@@@@@	000000	026056	000010	027056	000014	047272	000000	047166	026507	000014
1441	@_@@@@@@@@@@@@@@@@@@	046337	000000	047136	047165	047126	000000	026056	000010	027056	000014
1451	@@@@@@@@<@@@<@@@>@@@	045006	065014	045007	065010	136007	063012	136207	060006	137006	063002
1461	>@@@@@@@@@@@@@@@@@@@	137206	047136	047165	047126	000000	027056	000010	045007	047136	047165
1471	@@@@@g@@@@@@@:@@@@@@	047126	000000	044347	000000	027056	000010	047272	000000	047166	026507
1481	@@@_@@@@@@@@@@g@@@@	000010	046337	000000	047136	047165	047126	000000	044347	000000	027056
1491	@@@:@@@@@@@@_@@@@@@	000010	047272	000000	047166	026507	000010	046337	000000	047136	047165
1501	@@@@@@@@@@@@@@@@@@@@	047126	000000	014006	063522	005004	000200	065552	015007	065554	063016
1511	@@@$@@@@@@@@@@@@@@@@	060076	177644	065536	063476	015007	065536	063462	115004	065470	014007
1521	:@@@@@@@@@J+@@@@^@@@	135074	000030	062054	023006	041003	165253	017074	000200	157203	062404
1531	@@@@b@@@@@@t@@@@@@@@	017004	047165	161227	051004	064402	062364	077377	051404	017004	000074
1541	@@@@@@@@@@@@@@:@@h@p	000002	047165	027006	017004	047165	045007	047165	135074	000350	067760
1551	@@@@@@J/@@@@^@@L@@@@	042005	023006	041007	165257	013074	000200	157203	062714	017004	047165
1561	@@@%@V@@7@@@@@@@:@@@	015007	065646	063726	073200	133405	115004	063520	065474	135074	000030
1571	@L@@@@@@@J+@@@ @@@@	062314	014007	017003	023006	041003	165253	117203	065640	015004	041007
1581	@@><@@@@@@@@@@@^@@L	051404	137274	000000	077777	061006	044107	114074	000020	157207	055714
1591	@!9@@@@@@@@@@@@@@@h	177774	134405	065406	017004	063402	047165	077000	047165	177546	000350
1601	@@@@@@@@@@@@@>@@K@@@	067612	042005	023007	027006	017074	000200	060276	015007	145504	017006
1611	@@@'@:@@@@@6@@@@@@@	117206	063740	065272	042207	014005	060266	047136	177444	047126	000000
1621	@@@@@@@@@@@@@@@@@:@	014006	065536	063476	015007	065536	063462	115004	065470	014007	135074
1631	@@@@@@@J+@@@V^@@@@@	000030	062054	023006	041003	165253	017074	000326	157203	062404	017004
1641	@@b@@@@@t@@@@@@@@@@	000446	161227	051004	064402	062364	077377	051404	017004	000074	000002
1651	@@@@@@@@@@:@@h@p@@	047165	027006	017004	047165	045007	047165	135074	000350	067760	042005
1661	@@@@J/@@@@^@@L@@@@@	023006	041007	165257	013074	000200	157203	062714	017004	047165	015007
1671	@%@V@@7@@@@@@:@@@@L	065646	063726	073200	133405	115004	063520	065474	135074	000030	062314
1681	@@@@@@@J+@@@ @@@@@@	014007	017003	023006	041003	165253	117203	065640	015004	041007	051404
1691	><@@@@@@@@@@@^@@L@!	137274	000000	077777	061006	044107	114074	000020	157207	055714	177774
1701	9@@@@@@@@@@@@:@@h@@	134405	065406	017004	063402	047165	077000	047165	135074	000350	067612

Appendix A

An example Modula-2 Program - The Resource file (FDUMP)


```
  1711   @@@@@@@@@@@@>@@K@@@@@   042005 023007 027006 017074 000200 060276 015007 145504 017006 117206
  1721   @`@:@@@@@6@@@@@@@@@@   063740 065272 042207 014005 060266 047136 047165 047126 000000 015007
  1731   @@@@@@Z@X@@@7@7@Z@@@   063506 014006 063534 155105 154104 073200 133404 133405 155004 064522
  1741   @@9@b@@@@@@@@@@@FD@@   014003 134505 161135 041007 023007 044103 024006 041004 143304 044104
  1751   HGX@@@Y@@@@@@@NF@@^@   144307 154203 041104 154404 044104 044107 044106 147306 044106 157204
  1761   @@^<@@@@@@@@@@@@@@@@   065014 157274 000000 000200 017005 063432 047165 051405 064424 062422
  1771   @@^@^@@@b@@@@@@@@@@@   074100 157204 157207 062004 161227 051005 017005 063402 047165 077000
  1781   @@@z=@@<@@@@@@@@@@@@   047165 065372 136407 107274 177777 177577 045007 000074 000002 047165
  1791   @@@@@@@@@@@@@@@@@@Z@   047136 047165 047126 000000 015006 063556 024007 063576 073200 155105
  1801   X@7@7@@@@@@@@@@@>@@@   154104 133405 133404 114005 064572 041007 044107 044106 137106 065406
  1811   @@@@b@@@@@;@b@@@@F@@   052004 064544 161237 044107 015003 135504 161114 023007 103306 035003
  1821   FF@@@@@@@@@@FE@@@@@@   143306 117203 044107 044106 033006 041003 143305 117203 062010 023006
  1831   @@^@@@@@@@@@C@@@@@@   000042 157203 051505 023006 044103 041107 107303 044105 065410 035007
  1841   Z@@@@@@@Z<@@@@@@@@@@   155205 051404 037005 035007 155274 000000 000200 027005 017004 063442
  1851   @@@!@@@@@@@<@@@@@@@@   047165 107374 000000 045206 063206 107274 177777 177577 045007 000074
  1861   @@@@@@@@=@@j@z@@@@@@   000002 047165 044106 044107 136407 060352 065772 077000 047165 047136
  1871   @@@@@@@@@@@@@@@@@@`   047165 047126 000000 025074 000000 000137 045207 063462 065010 075340
  1881   @@@@@@><@@@@@@@@@@@@   042207 064450 051405 137274 000000 077777 061006 044107 115074 000020
  1891   ^@@M@!@@@@^<@@@@@@@b@   157207 055715 177774 045007 065016 157274 000000 000400 062006 161227
  1901   Z@@@@@@@@@@@@@@@@@@@   155074 000001 017005 047165 047136 047165 047126 000000 015007 065444
  1911   @@@@@@@@@@@@@@@@@j/   063422 041007 115074 000101 065426 115074 000037 065006 042005 165257
  1921   @@@@b@@@@@@@@@@@@@@   047165 077377 161217 000074 000002 047165 077000 047165 041007 115074
  1931   @A@t@@@@@@@@j/@@@@@@   000301 065764 115074 000037 065010 042005 165257 042207 047165 063006
  1941   @@@@@R@@@@@@@@@@@@@   042207 045207 065722 077000 004307 000037 000074 000002 047165 047136
  1951   @@@@@@@@@@@@@@@@@@@@   047165 047126 000000 047136 047165 047126 000000 000174 003400 070377
  1961   @@@@@@@@@@@@@@@@@@@@   020157 000002 021174 000000 000000 047321 047136 047165 047126 000000
  1971   @@@@@~@@@@@@@@@@@@@@   000174 003400 070376 020157 000002 021174 000000 000000 047321 047136
  1981   @@@@@@@@@@@}@@@@@@@@   047165 047126 000000 000174 003400 070375 020157 000002 021174 000000
  1991   @@@@@@@@@@@@@@@@@!@@   000000 047321 047136 047165 047126 000000 000174 003400 070374 020157
  2001   @@@@@@@@@@@@@@@@@@@@   000002 021174 000000 000000 047321 047136 047165 047126 000000 000174
  2011   @@@<@@@@@@@@@@@@@@@@   003400 070373 020157 000002 021174 000000 000000 047321 047136 047165
  2021   @@@@@@@@@@z@@@@@@@@@   047126 000000 000174 003400 070372 020157 000002 021174 000000 000000
  2031   @@@@@@@@@@@@@@@@y@@@@   047321 047136 047165 047126 000000 000174 003400 070371 020157 000002
  2041   @@@@@@@@@@@@@@@@@@@@   021174 000000 000000 047321 047136 047165 047126 000000 000174 003400
  2051   @x@@@@@@@@@@@@@@@@@@   070370 020157 000002 021174 000000 000000 047321 047136 047165 047126
  2061   @@@@@@@@w@@@@@@@@@@@@   000000 000174 003400 070367 020157 000002 021174 000000 000000 047321
  2071   @@@@@@@@@@!@@@@@@@@!@@   047136 047165 047126 000000 020774 000000 000000 000010 020774 000000
  2081   @@@@@@!@@@@@@@@!@@@@@@   000000 000014 020774 000000 000000 000020 020774 000000 000000 000024
  2091   @!@@@@@@@@!@@@@@@@@!@@   020774 000000 000000 000030 020774 000000 000000 000034 020774 000000
  2101   @@@@@@!@@@@@@@@!@@@@@@   000000 000040 020774 000000 000000 000050 020774 000000 000000 000054
  2111   @!@@@@@B@@@@@$!@P!@$   020774 000000 000000 000270 020074 000000 000244 120436 150374 000244
  2121   @!@@@@@@@@@@@@@@@@@@   110374 000064 031174 000000 020511 000004 020574 177777 177777 000010
  2131   @@@h@@@@@@@@@Y@!@@@@   041220 042750 000020 021174 000000 000000 032331 063374 021110 070147
  2141   @@@H@!@@@X@(@T@@@L@@   041041 050710 177774 020511 177730 041250 177724 020510 177714 022174
  2151   @@@@@@@@@@@N@@@@@@@:   000000 000000 020512 177630 024110 116716 027111 047136 047165 047272
  2161   @@@:q@)t@@@@@@@@@@@@   005472 047272 170474 124764 000000 000000 000000 000000 000000 000000
  2171   @@@@@@@@@@@@@@@@@@@@   000000 000000 000000 000000 000000 000000 000034 000062 000000 041517
  2181   @@@@@@@@@@@@@@@@@@@@   042105 000001 000012 000000 177777 000000 000400 000000 000000 000000
  2191   @@@@@@@@@@             177777 000000 001070 000000 000000
```

FILE CONTAINS 2195 WORDS

Appendix A

An example Modula-2 Program - The Resource file (Resdump)

```
TUE, 07 OCT 1986                    1                          listidy


OK, seg resdump
Enter resource file name : test.res
----- Resource Header -----
offset to resource data         :       256
offset to resource map          :      4328
total length of resource data   :      4072
length of resource map          :        28
directory entry details         : not output
user data :


----- Data -----
Block # 0
length of resource data block :        24 000030

 000000  :  000000  000050  000000  000160  000000  000010  000000  000040
 000020  :  000000  036474  000000  124760

----- Data -----
Block # 1
length of resource data block :      4036 007704

 000000  :  000000  000001  047372  007670  047126  177776  035574  000001
 000020  :  177776  023154  177774  033574  000002  177776  023154  177774
 000040  :  035053  177776  161505  023154  177774  034053  177774  044304
 000060  :  104774  000005  155104  023154  177774  033505  177774  023154
 000100  :  177774  035055  177776  155153  177776  023154  177774  155153
 000120  :  177774  036505  177776  047136  047165  047126  177774  027454
 000140  :  177774  024516  177774  035574  000001  177776  036574  000002
 000160  :  177776  036574  000003  177774  047272  000000  024537  177774
...
                   000000  000000  000040  020774  000000  000000  000050
 007520  :  020774  000000  000000  000054  020774  000000  000000  000270
 007540  :  020074  000000  000244  120436  150374  000244  110374  000064
 007560  :  031174  000000  020511  000004  020574  177777  177777  000010
 007600  :  041220  042750  000020  021174  000000  000000  032331  063374
 007620  :  021110  070147  041041  050710  177774  020511  177730  041250
 007640  :  177724  020510  177714  022174  000000  000000  020512  177630
 007660  :  024110  116716  027111  047136  047165  047272  005472  047272
 007700  :  170474  124764

----- Resource Map -----
  reserved - copy of header        : (16 bytes)
  reserved - handle to next rmap   : ( 4 bytes)
  reserved - file reference number: ( 2 bytes)
  resource file attributes         :        0
  offset to type list              :       28
  offset to resource name list     :       50
  number of resource types - 1     :        0
  --- type list ---
    resource type        :CODE
    number of these - 1  :        1
    offset to ref list   :       10
    --- ref list ---
    resource ID          :        0
    offset to res. name  :       -1
    resource attributes  :        0
    offset to data       :      256
    handle/res. spec     :        0
    resource ID          :        0
    offset to res. name  :       -1
    resource attributes  :        0
    offset to data       :      568
    handle/res. spec     :        0
```

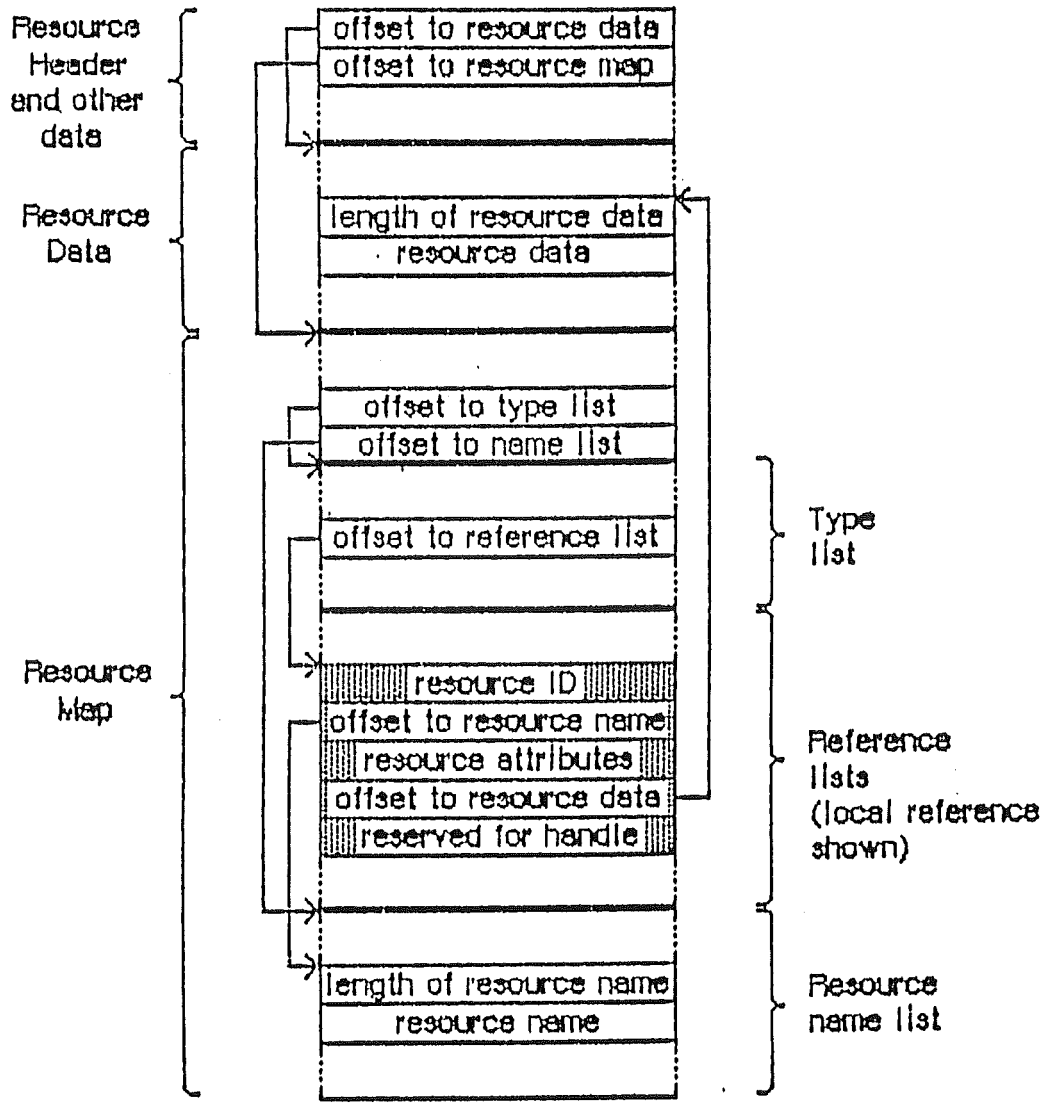
Appendix B

Resource file format - Summary

Resource Header and other data	4 bytes	Offset to resource data
	4 bytes	Offset to resource map
	4 bytes	Length of resource data
	4 bytes	Length of resource map
	112 bytes	Partial copy of file's directory entry
	128 bytes	User data
Resource Data	For each resource:	
	4 bytes	Length of following resource data
	n bytes	Resource data for this resource.
Resource Map	16 bytes	Reserved for copy of resource header
	4 bytes	Reserved for handle to next resource map to be searched
	2 bytes	Reserved for file reference number
	2 bytes	Resource file attributes
	2 bytes	Offset to type list
	2 bytes	Offset to resource name list
Type list	2 bytes	Number of resource types minus 1
	For each type:	
	4 bytes	Resource type
	2 bytes	Number of resources of this type minus 1
	2 bytes	Offset to reference list for this type
Reference lists (one per type, contiguous, same order as in type list)	For each reference of this type:	
	2 bytes	Resource ID
	2 bytes	Offset to length of resource name or -1 if none
	1 byte	Resource attributes
	3 bytes	If local reference, offset to length of resource data If system reference, Ø
	4 bytes	If local, reserved for handle to resource If system, resource specification for system resource: in high-order word, resource ID; in low-order word, offset to length of resource name or -1 if none
Resource name list	For each name:	
	1 byte	Length of following resource name
	n bytes	Characters of resource name

Appendix B

Resource file format - Local reference



Bibliography

1. *Programming in Modula-2*,
Niklaus Wirth, Springer-Verlag, 1982.

2. *Prime System Architecture Reference Guide (DOC3060-1SSF)*
Prime Computer Inc.

3. *Programming the Macintosh in Assembly Language*
James W. Coffron, Sybex Inc, 1986.

4. *Macintosh Revealed (Volumes One and Two)*,
Stephen Chernicoff, Hayden Book Co., 1985

5. Documentation on the Smiler-2 Modula-2 Cross-Compiler,
Institut for Informatik, ETH, Zurich.